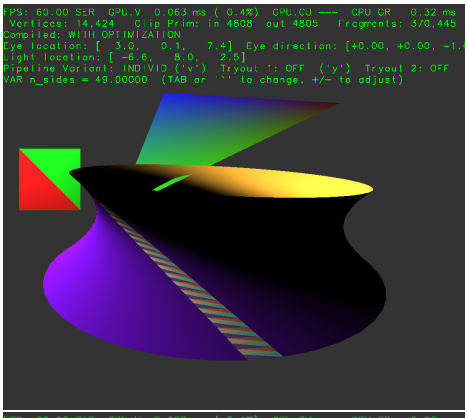
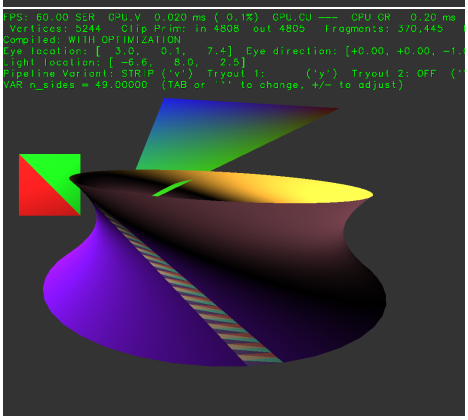
LSU EE 4702-1 **Homework 4** **Due: 14 November 2024**

All of the code for this assignment is in the course repo. HTMLized versions of the assignment file are at `https://www.ece.lsu.edu/koppel/gpup/2024/hw04.cc.html`.





Problem 0: If not already done, follow the instructions on `https://www.ece.lsu.edu/koppel/gpup/proc.html` for account setup and programming homework work flow. Compile and run the homework code unmodified. The homework code should initially show a scene with a rotating hyperboloid of one sheet, see the screenshot on the upper right. The screenshot on the lower right shows a correct solution to this assignment, in particular, the hyperboloid is rendered using a triangle strip (notice the vertex count) and it can be illuminated from behind.

Homework Overview

The code in this assignment explores methods of drawing a hyperboloid of one sheet in Vulkan. The baseline code in the unmodified assignment draws the hyperboloid using individual triangles.

User Interface

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Press `F12` to generate a screenshot. The screenshot will be written to file `hw04.png` or `hw04-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw04-1.ogg` or `hw04-debug-1.ogg`.

Initially the arrow keys, `PageUp`, and `PageDown`, can be used to move around the scene. Using the `Shift` modifier when pressing one of these keys increases the amount of motion, using the `Ctrl` modifier reduces the amount of motion. Use `Home` and `End` to rotate the eye up and down, use `Insert` and `Delete` to rotate the eye to the sides.

After pressing `l` the motion keys will move the light instead of the eye, after pressing `b` the motion keys will move the head ball around, and after pressing `e` the motion keys operate on the eye.

The simulation can be paused and resumed by pressing `p` or the space bar. Pressing the space bar while paused will advance the simulation by 1/30 s. Gravity can be toggled on and off by pressing `g`.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the gravitational acceleration, dynamic friction, and variables that will be needed for this assignment.

The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text. Pressing `Tab` and `Shift-Tab` cycles through the different variables. To locate variables which can be set this way, and to see how they were set, search for `variable_control.insert` in the assignment file.

Assignment-Specific User Interface

Pressing `1` generates a new hyperboloid. The first two are hard-coded, and after that they are randomly generated. Pressing `2` shows the first hyperboloid again.

Pressing `v` switches between using a pipeline using a triangle list (individual triangles) and a triangle strip. The setting is shown in the second line of green text to the right of label Pipeline Variant.

Display of Performance-Related Data

The top green text line shows performance in various ways. The number to the right of FPS shows the frame rate, the number of frames completed per second. On some displays 60 is the target frame rate and anything significantly lower than that indicates mediocre performance. Next, the green text shows whether frames are being prepared one at a time (serially), SER, or the steps in preparing a frame are being overlapped, OVR. In overlap mode commands for one frame are recorded while commands for a prior frame may be executing. For assignments early in the semester the mode will be kept at SER.

GPU.V shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows the execution of CUDA code per frame. CUDA code is physics in some assignments, but not this one and so the time should be shown as `---`. On some of the lab computers the computational accelerator GPU is different than the one performing graphics. CPU GR is the amount of time that the CPU spends recording Vulkan graphics commands (or whatever it does in the callback installed by `vh.cbs_cmd_record.push_back`). CPU PH is the amount of time that the CPU spends on physics or whatever it does in the callback installed by the call to `vh.display_cb_set`.

For rasterization the second line, the one starting with Vertices, shows the number of items being sent down the rendering pipeline per frame. Clip Prim shows the number of primitives before clipping (in) and after clipping (out). The next line indicates whether the code was compiled with optimization. Use the version without optimization for debugging and the version with optimization for performance measurements.

Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw04`, and a debug-able version of the code, `hw04-debug`. The `hw04-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw04-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page. **You must learn how to debug.** If not, you will be at a severe disadvantage.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In most assignments these include the variables `opt_tryout1`, `opt_tryout2`, `opt_tryout3`, and `opt_tryoutf`, available in CPU and shader code. You can use these variables in your code (for example, `if ( opt_tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification. Also see 2014 Homework 4-6 (especially 6), the spiral, to see examples of how the pipeline input topology can be chosen to provide data needed by the geometry shader, even though the topology class (point, line, triangle) does not geometrically correspond to what is being drawn.

Problem 1: The code in `World::render` prepares Vulkan for rendering a scene including some triangles, and a hyperboloid of one sheet. It also writes the green text. Two pipelines are used to render the hyperboloid, `pipe_hyperb` and `pipe_hyperb_strip`. Other than the primitive topology, both are identical.

The code computes hyperboloid vertex coordinates and normals and writes them to host containers `coords` and `norms`:

```
for ( int side = 0; side <= n_sides; side++ ) {
    float theta = side * delta_theta;
    pMatrix_Rotation rot(hyperb_az,theta);
    pCoor line_bot_pos = bot_ctr + rot * bot_vx;
    pVect l_dv = rot * line_dv;

    for ( int lev = 0; lev <= n_levels; lev++ ) {
        pCoor p_line = line_bot_pos + lev * l_dv;
        coords[lev][side] = p_line;
        pCoor ctr = bot_ctr + lev * delta_z;
        pVect tc = cross(hyperb_az,pVect(ctr,p_line));
        pNorm n = cross(tc,l_dv);
        norms[lev][side] = n;
      }
  }
```

Variable `lev` indicates the level (position along the hyperboloid axis) and `side` indicates a position around the circle at a level. Both are integers and are used to index the container. (See the complete code in the assignment file, and if necessary review Homework 2 which also worked with hyperboloids.) Notice that the `lev` loop moves along a straight line and that each iteration of the `side` loop rotates that line around the hyperboloid axis.

Continued on next page.

Variable `pipeline_variant` indicates which pipeline and vertex grouping to use. Pressing `v` toggles it between `PV_Individ` (triangle list [individual triangles]) and `PV_Strip` (triangle strip). Next, the code prepares a buffer set for the appropriate pipeline. In the unsolved assignment only individual triangles will work correctly.

```
switch ( pipeline_variant ) {
case PV_Individ:

  bset_hyperb.reset(pipe_hyperb);
  for ( int side = 0; side < n_sides; side++ )
    for ( int lev = 0; lev < n_levels; lev++ )
      {
        pCoor p00 = coords[lev][side];
        pCoor p01 = coords[lev][side+1];
        pCoor p10 = coords[lev+1][side];
        pCoor p11 = coords[lev+1][side+1];

        bset_hyperb << p00 << p01 << p10;
        bset_hyperb << p01 << p11 << p10;

        pNorm n00 = norms[lev][side];
        pNorm n01 = norms[lev][side+1];
        pNorm n10 = norms[lev+1][side];
        pNorm n11 = norms[lev+1][side+1];

        bset_hyperb << n00 << n01 << n10;
        bset_hyperb << n01 << n11 << n10;

        int c1 = side == 0 ? ( lev%2 ? 5 : 2 ) : 1;
        int c2 = side == 0 ? ( lev%2 ? 3 : 4 ) : 1;

        bset_hyperb << c1 << c1 << c1 << c2 << c2 << c2;
      }
break;
```

The code above inserts coordinates, normals, and color indices into buffer set `bset_hyperb` in the correct order for individual triangles. The color indices, `c1` and `c2`, are integers. They refer to colors stored in array `uni_hw03_colors`, which is prepared in `World::render`. A single index, such as 1, specifies both a front color and a back color. Though a different color index can be assigned to each vertex of a triangle, the geometry shader only uses the color index of the third vertex for coloring.

In the unsolved assignment the `PV_Strip` is identical to the code under `PV_Individ`. As a result the triangle strip pipeline will be fed with vertices in individual triangle order. As a result coloring will not be correct and the number of vertices will be 3× what they should be.

Modify the `PV_Strip` case in the `switch` (not shown above) so that vertices are inserted in the correct order for a triangle strip. In a correct solution the version using triangle strips will be identical in appearance to the one using a triangle list, but using one third the number of vertices.

To do this correctly one must use multiple triangle strips, either one per value of `side` or one per value of `lev`. When starting a new triangle strip (*restarting* a strip) use 0 for the color index of

the first two vertices. If triangle strips are not restarted properly extra triangles will be rendered, affecting at least coloring.

Problem 2: In the unmodified assignment the fragment shader, `fs_main` in file `hw04-shdr.cc`, applies a simple lighting algorithm that uses a diffuse coloring model to compute the lighted color of the fragment. First, the shader computes the value of `attenuation` which indicates how much light is reaching the fragment based on the distance from the fragment to the light and the angle, and on whether the light is on the side of the triangle that we can see (variable `lit_side` is `true`). If the light is on the opposite side of the triangle ((variable `lit_side` is `false`) then attenuation is set to zero. The shader then combines the visible color material property, light color, and attenuation to compute the lighted color:

```
void fs_main() {
  // Vector from fragment to light.
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;

  // Distance squared.
  float dist_sq = dot( vec_vl, vec_vl );

  // False if the light illuminates the side we can't see.
  bool lit_side = dot( normal_e, vec_vl )>0 == dot( normal_e, -vertex_e.xyz )>0;

  // Amount of light reaching the fragment.
  float attenuation = lit_side ? abs( dot( normal_e, vec_vl ) / dist_sq ) : 0;

  // Material color.
  vec4 mat_color = gl_FrontFacing ? uc.front[color_idx] : uc.back[color_idx];

  // Compute lighted color.
  frag_color = mat_color * uni_light.color * attenuation;
}
```

The lighting code above models an opaque material. Modify the code above to modify a translucent material as follows: If the lit side of the fragment is visible then compute the lighted color as shown above (and in the unmodified assignment). If the lit side is not visible then combine the front and back colors in some way so that the visible color is a combination of the two, and is darker than if the light were on the visible side. See the lower screenshot on the first page of this assignment.